UNIVERSITAT POLITECNICA DE
CATALUNYA

PARALLELISM

Lab 2: Brief tutorial on OpenMP programming model

Roger Vilaseca Darne and Xavier Martin Ballesteros
PAR4110

20th March 2019, Q1

Contents

1 Introduction

Talk about the Mandelbrot Set and about what we are going to do in this 3 sessions, in order.

2 Task decomposition analysis with *Tareador*

In this section, we had to analyse the two possible task granularities that could be exploited in the given program. To do it, we used the *Tareador* tool, which was very useful to see graphically the created tasks and which dependences are between them.

2.1 Point decomposition

In this decomposition strategy, a task corresponds with the computation of a single point (row, col) of the Mandelbrot set. Thus, we will have $row \times col$ tasks.

In order to analyse the potential parallelism of this strategy, we modified the code in *mandel-tar.c* to create several *Tareador* tasks. As we are using a point decomposition strategy, the tasks are created inside the most inner loop of the function mandelbrot. Figure 1 shows the fragment of the function that we modified.

```
for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {
        tareador_start_task("point");

        ...

        tareador_end_task("point");
    }
}
```

Figure 1: Modified fragment of the *mandel-tar.c* code.

Once we did this, we executed interactively mandeld-tar and mandel-tar using run-tareador.sh script. The first one is used for timing purposes and to check for the numerical validity of the output (-o option) whereas with the second one we can visualize the Mandelbrot set.

The script has defined inside the size of the image to compute (-w option). In this case, the value was 8 to generate a reasonable task graph in a reasonable execution time. Hence, as we said before, the number of tasks will be $8 \times 8 = 64$.

Figures 2 and 3 below show the two task decomposition graphs (TDG) of the two possible executions. There are some nodes bigger than the others. This is because these nodes have executed more instructions than the rest. Therefore, these nodes represent pixels in white[1].

[1]The bigger the node, the more iterations it does in the do while fragment of the function. This make that the computed color is white.

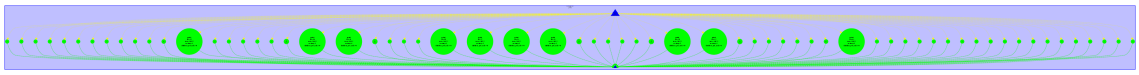

Figure 2: Mandel-tar task decomposition graph using the point decomposition strategy.

Figure 3: Mandeld-tar task decomposition graph using the point decomposition strategy.

This strategy will have more overhead of creation and termination of tasks than in the row strategy because it has to create more tasks. However, as a positive point the tasks are better distributed because if in a row there are many white areas, this work will not only be done by a single thread. Thus, it may end the execution earlier than in the row decomposition strategy.

2.2 Row decomposition

In this other strategy, a task corresponds with the computation of a whole row of the Mandelbrot set. This strategy only creates *row* tasks.

In this case, the code is not the same as before. We changed the creation of the *Tareador* tasks so that each time we enter a new row (second loop) a new task is created. The modified version of the code is shown below.

```
for (row = 0; row < height; ++row) {
    tareador_start_task("row");
    for (col = 0; col < width; ++col) {
        ...
    }
    tareador_end_task("row");
}
```

Figure 4: Modified fragment of the *mandel-tar.c* code.

Afterwards, we executed interactively mandel-tar and mandeld-tar again. We used the same size as before. Hence, the total number of tasks will be 8. The graphical results we got can be seen in figures 5 and 6.

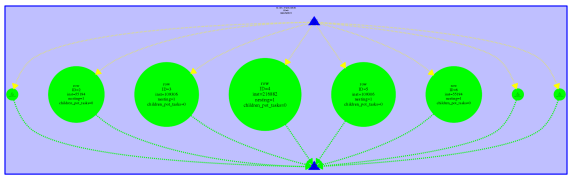

Figure 5: Mandel-tar task decomposition graph using the row decomposition strategy.

Figure 6: Mandeld-tar task decomposition graph using the row decomposition strategy.

In this strategy, the overhead time of creation and termination of tasks will be very small in comparison with the point strategy because it has to create less tasks. Nevertheless, it may happen that in a full row, all of its pixels must be white. In this case, it can happen that while other threads have finished their work, this other thread is still executing the row. Consequently, it is possible that the execution time may be bigger than in the point decomposition strategy.

2.3 Characteristics of the TDG

We saw in the previous sections that the execution of the mandel-tar has a very different task dependence graph than the execution of the mandeld-tar.

On the one hand, we can see that in mandel-tar every point is independent from the others. Consequently, we could parallelize that fragment of the code using OpenMP clauses.

On the other hand, in mandeld-tar we can see that all iterations have became sequential. In this situation, we do not gain anything by parallelizing the code, but we increase the execution time because of the overhead of creation and termination of tasks.

Using the *Tareador* we could see which variable was responsible of creating those dependences. We did the following: Right Click into a task -> Data View -> Edges-out -> Real Dependency. Figure 7 shows the result we got.

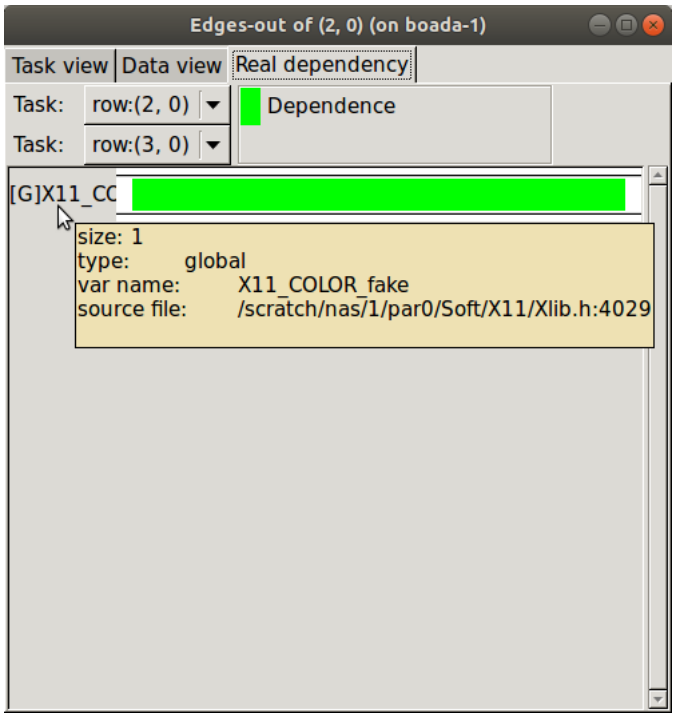

Figure 7: Variable that provoques the dependences between tasks.

We can see that there is only one variable that is causing all the dependences: X11_COLOR_fake. Observing the code, we noticed that the only difference between mandel-tar and mandeld-tar was a fragment of the code that was only executed in mode _DISPLAY_ (mandeld-tar):

```
#if _DISPLAY_
    /* Scale color and display point  */
    long color = (long) ((k-1) * scale_color) + min_color;
    if (setup_return == EXIT_SUCCESS) {
        XSetForeground (display, gc, color);
        XDrawPoint (display, win, gc, col, row);
    }
#else
    output[row][col]=k;
#endif
```

Figure 8: Fragment of the *mandel-tar.c* code.

Therefore, variable X11_COLOR_fake is used at least in one of the functions XSetForeground and XDrawPoint. We could protect this section of code in the parallel OpenMP code using the *critical* clause to define a region of mutual exclusion where only one thread can be working at the same time.

```
#if _DISPLAY_
    /* Scale color and display point  */
    long color = (long) ((k-1) * scale_color) + min_color;

    #pragma omp critical
    if (setup_return == EXIT_SUCCESS) {
        XSetForeground (display, gc, color);
        XDrawPoint (display, win, gc, col, row);
    }
#else
    output[row][col]=k;
#endif
```

Figure 9: Fragment of the *mandel-tar.c* code using the *critical* clause to protect a fragment of the code.